Dispatching Agents in Electronic Institutions

Hector G. Ceballos
Tecnologico de Monterrey
Ave. E. Garza Sada 2501
Monterrey, Mexico
ceballos@itesm.mx

Pablo Noriega
IIIA-CSIC
UAB Campus
Bellaterra, Spain
pablo@iiia.csic.es

Francisco J. Cantu
Tecnologico de Monterrey
Ave. E. Garza Sada 2501
Monterrey, Mexico
fcantu@itesm.mx

ABSTRACT
In Electronic Institutions [1], agents may be prevented from achieving their goals if other participants are not present in a given scene. In order to overcome this situation we propose the addition of an institutional agent in charge of dispatching agents to scenes through a participation request protocol. We further propose to endow this agent with the capability of instantiating new agents, thus providing grounds for a self-optimization of the system. Advantages of our proposal are illustrated with the implementation of an information auditing process.

Categories and Subject Descriptors
I.2.11 [Distributed Artificial Intelligence]: Multiagent systems

Keywords
Multiagent systems development environments. Electronic institutions. EIDE.

1. INTRODUCTION
The Electronic Institutions framework [1] developed in the Artificial Intelligence Institute of the Spanish National Scientific Research Council, (IIIA-CSIC), is a means to design and implement regulated open multiagent systems. The current framework is the outcome of more than a decade of developments and has been used to implement regulated MAS in several domains. For example, to support electronic auctions, to establish supply-chain virtual organizations, to agentify hotel and hospital management systems, to support participative experimentation, to model policy-making or to simulate human activity in archaeological sites. The framework has also served as a model-building environment for the discussion of topics like agent-based simulation, machine readable normative languages or autonomic computing. But in spite of this considerable variety of modes of use there have been very few published references to the underlying technology and, in particular, seldom any discussion of its expressive limitations and ways to circumvent them [2, 4].

This paper is one such discussion. We address the problem of deadlocks induced by improper institutional support in the follow-through of processes whose “most natural” representation may be as goal-directed workflows. In fact, we frame that problem in slightly more general terms: as the breakdowns produced by stalling or absent agents; and we advance a solution whose schema—an institutional agent with particular functionalities—may be reused mutatis mutandis to address similar problems and may also be coded as a standard functionality in the framework infrastructure. We believe that our solution should facilitate the adoption of the electronic institutions metaphor for the design of conventional MAS.

The structure of the paper is straightforward. The next subsections provide terminological and conceptual background. In Sec. 2 we present our proposal and in Sec 3 we describe a case study on information auditing to illustrate our proposal and present simple experimental results to back our claims. We finish with a brief discussion and comments on future work.

1.1 Electronic Institutions
For the purpose of this paper the EI framework may be described in terms of a conceptual model, a computational model and a software platform, EIDE, to specify and run electronic institutions [1, 5].

The conceptual model for electronic institutions assumes that the electronic institution determines a virtual space where agents interact subject to explicit conventions, so that institutional interactions and their effects count as facts in the real world. Because of this virtuality, it is assumed that all interactions within the electronic institution are speech acts expressed as illlocutionary formulae. The electronic institution defines an open MAS in the sense that (i) it makes no assumption about the architecture and goals of participating agents (who may be human or software entities); and (ii) agents may enter and leave the institution at will, as long as the regimented conventions of the institution are met. Participating agents are subject to role-based regulations whose specification is given in terms of illocutions, norms and protocols. There are two classes of agents, internal and external. Internal agents act on behalf of the institution itself who is responsible for their behavior. External agents act on their own behalf and their internal state is inaccessible to the institution. Interactions are organized as repetitive activities called scenes. Scenes establish interaction protocols describing agent group meetings as transition diagrams whose arcs are labeled by valid illocutions. The performative structure captures the relationships among scenes describing those transitions agents playing certain role can make. Finally, normative rules describe the obligations an agent contracts while it participates in the institution. Agents may move from one scene to another, they may be active in more than one scene at a given time and they may perform different roles in different scenes.

The computational model for EIs defines a social (in-
A thorough review of how stalling and deadlocks are addressed in other platforms is beyond the scope of this paper; here we merely point towards the grounds that are common to most approaches and two environments that use a notion of interaction context assimilable to EI scenes.
in order to reach a common goal. Similarly, in AGR (Madkit) agents organize themselves in groups adopting one role and are limited to communicate only with other agents in the same group. Nevertheless, an agent can join multiple groups simultaneously. In Electronic Institutions, once inside a scene, agents are allowed to exchange messages only with other agents in that scene.

The three approaches propose a common space or context on which agents interact in order to achieve a single common goal, or multiple individual ones. Coordination requires controlling the entrance and exit of participants as well as the minimum/maximum number of agents playing certain role. Agents are free to decide which space(s) to join as their own goals dictate. However, none of the three approaches provides a solution for those cases when there are not enough agents for starting a group or scene, or when a group or scene is stalled waiting for one or more agents to continue. Agents in the stalled group need to communicate with other agents in order to invite them to join.

Our approach proposes the existence of a Dispatcher Agent in charge of directing the invitations made by agents in stalled groups/scenes. This agent optimizes the allocation of resources instantiating new agents when necessary and negotiating with available agents their participation.

2. REQUESTING AGENT PARTICIPATION

The context established by a scene in Electronic Institutions makes it difficult for agents in the scene to reach other agents. Such confinement creates a challenge: to warrant that all required agents be present in the scene. We formalize this problematic situation and propose a solution. We propose to institute an agent in charge of dispatching agents to scenes through a participation request protocol. This agent makes use of the low-level services recommended by FIPA and is coherent with the EI model and implementable in the current EIDE version.

2.1 Missing agents in scenes

Assuming that agents decide freely to enter or not in a scene, we may find two problematic situations in which participants would not achieve their individual goals: 1) not all the agents required for the scene are available, or 2) an agent in the institution is not aware that its participation is required in a particular scene. Both conditions can appear before the creation of the scene or during its execution. In the following we will only deal with the case of ongoing scenes.

Formally, the problematic situation would be defined as follows. There is an agent \( A_1 \) pursuing a goal \( G_1 \) that is currently playing role \( R_1 \) in scene \( S \). Scene \( S \) is in state \( W_k \) and the achievement of \( G_1 \) requires reaching state \( W_n \). To do so, there is a sequence of illocutions \((M_1, \ldots, M_n)\) that must be issued by \( A_1 \) or by some other agent \((A_2, \ldots, A_n)\) playing roles \((R_2, \ldots, R_n)\) in \( S \). Nevertheless, at state \( W_j \) the outgoing illocution \( M_j, \ 1 \leq j \leq n \), has for sender or receiver an agent playing role \( R_j \) for which there is no agent in the scene. We assume that there exists a state \( W_k, \ 1 \leq k \leq j \), at which the entrance of agents playing role \( R_j \) is allowed and in which \( A_1 \) is capable of keeping the scene on hold.

In order to reach \( W_n \), agent \( A_1 \) sets \( \text{meansFor}(G_2, G_2) \) and \( \text{meansFor}(G_3, G_1) \), where \( G_3 = \{ \text{holdAt}(S, W_k) \} \) and \( G_2 = \{ \text{agentsPlayingRole}(R_j, S, Q) \} \). \( \text{meansFor}(G_2, G_1) \) denotes that goal \( G_1 \) is in stand-by until \( G_2 \) is achieved. Likewise, \( \text{holdAt}(S, W) \) is a goal that is satisfied when scene \( S \) reaches state \( W \). Similarly, \( \text{agssPlayingRole}(R, S, Q) \) is satisfied once there are \( Q \) agents playing role \( R \) on scene \( S \), where the quantifier \( Q \in \{ \text{ONE, ALL, 8..n} \} \), represents only one agent, any available agent, or exactly \( n \) agents, respectively.

In order to achieve the goal \( \text{agssPlayingRole}(R, S, Q) \), the agent \( A_1 \) must request the participation of other agents through some protocol \( P \). Such protocol \( P \) must achieve the agreement of \( Q \) agents to participate in \( S \) with role \( R \). Protocol \( P \) might include agent selection and negotiation, that may be performed by \( A_1 \) itself or by another agent.

2.2 A Dispatcher Agent

We introduce the notion of Dispatcher agent as an intermediary agent that facilitates the achievement of those \( \text{agssPlayingRole}(R, S, Q) \) goals owned by other agents. Let us represent the Dispatcher agent with the symbol \( A_D \) and denote its attributions with the role \( R_D \). This agent keeps track of all agents on the institution through the Agents relation. Besides, \( A_D \) maintains the three following relations: \( \text{AgClasses}, \text{hasType} \) and \( \text{canPlay} \). The set of agent classes \( \text{AgClasses} = \{ C_1, \ldots, C_n \} \) represent the software implementation of any participant, denoted by a source code class. Through \( \text{hasType} \), \( A_D \) keeps track of the agent class of every agent in the institution; it is assumed that every agent belongs to a single agent class. Finally, \( \text{canPlay} \) is used to know which roles may be played by an agent according to its agent class. The \( \text{canPlay} \) set may be built and updated by keeping track of participants in the institution, or may be known a priori. \( A_D \) is capable of creating new instances of the agent class \( C_i \) through the action \( \text{Instantiate}(C_i) \), which creates and enters an agent \( A_i \) in the institution. The configuration of \( A_D \) specifies for each agent class, a maximum number of agents it can manage, denoted \( \text{MaxAgs}(C_i) \). If \( \text{MaxAgs}(C_i) \) is 0, \( A_D \) cannot instantiate agents of type \( C_i \). The function \( \text{CurrAgs}(C_i) \) counts the number of tuples \( \{AG_i, C_i \} \in \text{hasType} \).

\( A_D \) implements two primitive operations for updating these relations. \( \text{RegisterAgent}(A_i, C_i) \) inserts \( A_i \) in \( \text{Agents} \) and \( \text{AgClasses} \), \( \text{hasType} \). On the other hand, \( \text{UnregisterAgent}(A_i, C_i) \) removes \( A_i \) from \( \text{Agents} \) and \( \text{AgClasses} \), \( \text{hasType} \). Similarly, \( \text{CanPlay}(A_i, C_i) \) set is used to know which roles may be played by an agent according to its agent class. The \( \text{CanPlay} \) set may be built and updated by keeping track of participants in the institution, or may be known a priori. \( A_D \) is capable of creating new instances of the agent class \( C_i \) through the action \( \text{Instantiate}(C_i) \), which creates and enters an agent \( A_i \) in the institution. The configuration of \( A_D \) specifies for each agent class, a maximum number of agents it can manage, denoted \( \text{MaxAgs}(C_i) \). If \( \text{MaxAgs}(C_i) \) is 0, \( A_D \) cannot instantiate agents of type \( C_i \). The function \( \text{CurrAgs}(C_i) \) counts the number of tuples \( \{AG_i, C_i \} \in \text{hasType} \).
to participate in $S$ playing role $R$. The acceptance of $AG_i$ is represented by $Agree(AG_i, S, R)$, while its refusal is represented by $Refuse(AG_i, S, R)$. Once all agents in AGS have given a response, $AD$ proceeds to select the best agents for the role.

The set of agents accepting the invitation, denoted $AccAgs$, is partially ordered by an operator $\geq$ that calculates how suitable is an agent $AG_i$ of type $C$ for playing role $R$ in scene $S$, denoted $AccAgs_{>\geq}$. Hence, $AG_i \geq AG_j$ means that $AG_i$ is better or at least as good as $AG_j$ for the given scenario.

The ordered set $SelAgs \subseteq AccAgs_{>\geq}$ is constituted by the first $n$ agents of $AccAgs_{>\geq}$, where $n = 1$ for $q = ALL$, $n = |AGS|$ for $q = ONE$, and $n \leq n$ for $Q = N^n$.

Definition 2. An apr $APR(X, R, S, Q)$ is satisfied, denoted $satisfied(apr)$, if $|SelAgs(apr)| \geq 1$ for $Q \in \{ONE, ALL\}$ or $|SelAgs(apr)| = n$ for $Q = N^n$.

If apr is not satisfied w.r.t. Agents, the introduction of new agents in the system would solve the problem. This is possible if there exists at least one agent class $C_i$ such that $canPlay(C_i, R)$ and $MaxAgs(C_i) > 0$. If there is no $C_i$ with these properties, $AD$ will have to wait for new agents for a fixed period of time $\tau$, after which it will declare the request unsatisfied.

Given that there may be more than one agent class capable of playing role $R$, $AD$ can use the same partial order criteria $\geq$ for selecting the best class for the role $R$ required in an apr. If apr is an agent participation request and $AgClss(apr)$ a partially ordered set $\{C_i|canPlay(C_i, R)\}$ w.r.t. $\geq$, then $AD$ will choose the first $C_i \in AgClss(apr)$ for which $CurrAgs(C_i) < MaxAgs(C_i)$. If no $C_i$ satisfies this requisite, the instantiation is not performed.

$AD$ determines the number of agents that should be instantiated to satisfy the request, denoted $NMissing$. If $Q = ONE, NMissing = n - |SelAgs|$, meanwhile if $Q \in \{ONE, ALL\}$ then $NMissing = 1$. If $NMissing > 0$, $AD$ can instantiate and enter in the institution a missing agent through the execution of the primitive $Instantiate(C_i) : A_i$ for some $C_i \in AgClss(apr)$. These primitives make use of the Agent Management System (AMS) provided by any FIPA-compliant agent platform.

Agents entering the institution are invited to scenes held in stand-by due to an unsatisfied apr. Thus, if an agent $AG_i$ of type $C_i$ is created by $AD$ in order to satisfy apr $APR(X, R, S, Q)$ and $AG_i$ doesn’t accept the corresponding invitation to $S$, $C_i$ is removed from $AgClss(apr)$. If an agent created by $AD$ refuses all the invitations made during its logging in the institution, its access is denied. Agents exiting from the institution produce a revision of unsatisfied agent participation requests that might require the instantiation of new agents.

We distinguish between permanent and transient participants according to their patterns of entry and exit in the institution. Let’s call permanent participants those agents that remain in the institution continuously while it is alive. On the other hand, transient participants are agents that enter the institution pursuing certain goals and exit once they have reached them. Agent classes representing permanent participants are identified by the set $PermAgClss \subseteq AgClasses$; similarly transient participants are denoted by $TransAgClss \subseteq AgClasses$.

Now we can establish necessary conditions to determine when an unsatisfied agent participation request justifies an agent instantiation.

Theorem 1. An unsatisfied apr $APR(X, R, S, Q)$ can be satisfied through the instantiation of $NMissing$ agents if there is a subset $(C_i \cup C_j) \subseteq AgClss(apr)$ such that $NMissing \leq |FSlots(apr)|$, where

$$FSlots(apr) = \sum_i MaxAgs(C_i) - CurrAgs(C_i, S) + \sum_i MaxAgs(C_j) - CurrAgs(C_j, S)$$

for $C_i \in \{AgClss(apr)\} \cap TranAgClss$ and $C_j \in \{AgClss(apr)\} \cap PermAgClss$. $CurrAgs(C_i, S)$ returns the number of agents with type $C_i$ currently in scene $S$.

Proof. Eventually, transient agents will leave the institution releasing slots that $AD$ can use for creating new instances, hence in the worst case where $MaxAgs(C) = CurrAgs(C)$ and a single $C \in AgClss(apr) \cap TranAgClss$ exists, the exit of all agents of type $C$ will make $CurrAgs(C) = 0$ allowing the instantiation of the required agents.

$CurrAgs(C, S)$ allows to consider those agents of class $C$ that will remain in $S$. For permanent agents, we cannot assume that they will exit from the institution, hence we can only count with the instantiation of $MaxAgs(C) = CurrAgs(C)$ agents of type $C$. \[\square\]

The order in which invitations are issued is important when incoming agents have a limited capacity for attending invitations. Suppose that $AD$ is processing two agent participation requests apr1 and apr2 for the same role $R$, where $AgClss(apr1) = AgClss(apr2)$, and the maximum number of invitations an agent of type $C$ in $AgClss(apr1)$ can take is one. If an agent is instantiated for class $C$ and the invitation for apr2 is sent earlier than the invitation for apr1, the new agent will only attend the scene in apr2. Similar instantiations and invitations might satisfy apr2 and left apr1 in hold if $NMissing_{apr1} < |FSlots(apr2)|$.

On the other hand, the refusal of agents for participating in an apr might produce an empty $AgClss(apr)$ set. This condition would allow $AD$ to consider apr unsatisfiable and discard it from its queue. Otherwise, an unsatisfiable apr1 might block a subsequent apr2 if $AgClss(apr2) \subseteq AgClss(apr1)$.

2.4 Request and Invitation Protocols

Agent participation is negotiated through two protocols, one for requesting agent participation ($P_{req}$) and another for inviting agents ($P_{inv}$). Both protocols must be executed in parallel with the scene that originated the request for agent participation.

Let us use the $DAgent$ name for denoting the $RD$ role, call $ReqAgent$ the role played by an agent requesting the participation of other agents and call $InvAgent$ the role that an invited agent plays. Every agent in the institution must be able to play $ReqAgent$ and $InvAgent$ roles, meanwhile only one agent, $AD$, is allowed to play the role $DAgent$.

Figure 1 shows the sequence diagram for $P_{req}$ between $DAgent$ and $ReqAgent$. Figure 2 depicts the automata describing the request protocol where letters on arrows represent valid sequences of illocutions taken from figure to ascribing the request protocol where letters on arrows representing the request protocol.
3. CASE STUDY

We used the Electronic Institution formalism for the automation of the auditing of an information repository. Several kinds of autonomous agents and human users participate in this auditing process. The process is initiated by external events and during its execution human intervention might be required. Rather than waiting for human users entry to the system, our approach enables autonomous agents to request human participation in order to achieve their goals.

A multiagent system for performing this auditing process was implemented with the tools developed in the IIIA [5]. Experiments and the results obtained are described at the end of this section.

3.1 Information Auditing

The information repository is managed by a RepGuardian agent that monitors changes on the repository and initiates the auditing process. The auditing process is driven by a specialized agent Carrier, and with the participation of other autonomous agents, Auditor and Corrector, as well as user agents representing human experts and information authors. A Carrier agent receives a notification about a record that has been added or modified in the information repository. The Carrier requests every available Auditor agent to check the internal consistency of the repository with respect to the auditing rules it knows. The Auditor agent responds to the Carrier whether informing that the record and the repository are consistent or returning a set of the inconsistencies detected. The type of inconsistencies are either internal inconsistencies of the record, or violations of rules defined for the entire repository; for instance, duplicity of records.

The Carrier agent chooses between sending the record to automatic correction with a Corrector agent, asking for expert assessment from a human expert, or notifying the author of the record of the possible inconsistency. The Corrector agent can apply the correction procedure or ask for expert assessment instead. In turn, the Expert user can modify the record or notify to its author. At the end, the decision made by the author is final. This decision model is depicted on Figure 5.

In this scenario we can detect some cases where human
users are not present in the system when their participation is required. For instance, the user registering or modifying the record might have left the system by the time an inconsistency needs a final decision. Similarly, the expert user may not be logged in the system when an inconsistency is detected. A new auditing rule evaluated throughout the repository might require the assessment of expert users or users responsible for inconsistent records.

3.2 Implementing the auditing process

The process described above was specified with a per-formative structure AuditingPS that contains protocols for: triggering the auditing (NewInfo), detecting inconsistencies (Auditing), performing the corrections (Correction) and integrating results on the repository (Audited). AuditingPS and its protocols use the roles defined above: RepGuardian, Carrier, Auditor, Corrector, Expert and Author. Using the ABuilder tool [5], agent classes were generated for each role, except for Expert and Author roles which shared the same agent class, named UserAgent. Only the RepGuardianAgent was classified as permanent; the rest of the agent classes were considered transient.

It was possible to simulate the auditing of new pieces of information registered in the repository with this implementation. The simulation required to have a fixed set of user agents playing the roles of experts and authors for every possible human user that could be required in the process. Besides, every expert or author user participated in each Correction scene.

Given that human users responses to a request made when he/she was off-line might take entire days, the duration of Correction scenes was limited by a timeout after which the correction is considered to have failed. A User agent representing an expert or an author is not allowed to participate on multiple scenes simultaneously; nevertheless it can accept invitations to other scenes until reaching a given limit of invitations.

3.3 Implementing Agent Participation Request

The request for agent participation was implemented by developing: 1) an additional per-formative structure containing the protocols proposed in our approach, 2) the dispatcher agent, 3) an institutional service for instantiating new agents, and 4) new functionality for previously defined agent classes.

The new per-formative structure is assembled with four protocols that enable agents to: 1) log in, 2) request agent participation, 3) receive and answer invitations to scenes, and 4) log out. AuditingPS was inserted in this per-formative structure indicating that every agent should pass by the first three protocols before entering AuditingPS, hence remaining active in request and invitation protocols. Finally, after leaving AuditingPS they should pass by the log out scene. Protocols and roles specified in this per-formative structure are defined in section 2.4.

The RepGuardian agent implemented the functionality of the DAgent role with the characteristics described in section 2.2, the algorithm outlined on section 2.3 and the primitives described in both sections.

Agents developed for AuditingPS were augmented with the functionality of ReqAgent and InvAgent roles. A parameter on each agent class C denoted MaxInv(C) was set to limit the maximum number of simultaneous invitations an agent of this class can accept.

3.4 Experiments

To demonstrate the capabilities of the Dispatcher agent we prepared a test-bed with the system described above. We want to observe the capabilities of the DAgent for dispatching agents to scenes where human intervention is requested and for detecting unsatisfiable requests. In order to do so, we simulated different demand patterns on the system and manipulated the maximum number of agents permitted. An overloaded system is that in which information is fed faster than users are able to revise it. Thus we provoked that certain scenes stalled due to the lack of enough agents for all of them. Next we manipulated the maximal number of agents in order to generate unsatisfiable requests.

We defined a single RepGuardian and constant populations of auditor and corrector agents. One Carrier agent was instantiated for each information piece fed into the system. User agents playing the role of Expert or Author are created on demand up to a maximum of MaxAgs(Carrier). The same User agent representing a human user must participate in all the scenes where the user intervention is requested and it must wait to finish its work in a scene before proceeding to the next.

Our focus was on the Correction protocol, whose decision model is shown in Figure 5 and is explained in section 3.2. In our experiments, the power for instantiating Corrector agents was disabled, i.e. MaxAgs(Corrector) = 0. In consequence, the dispatcher informs the Carrier of the unfeasibility of requests for Corrector agents. Hence the Carrier requests an expert who in turn calls one author for correcting the record. In conclusion, every Correction scene is initiated by one Carrier and requires the participation of one expert and one author. All the agents remain in the scene until this finishes. User agents were limited to accept up to three invitations, i.e. MaxInv(User) = 3.

We prepared three system configurations. The first configuration gives us a reference of how the system would behave under low demand. In the second we have an information feeding rate higher than revision time, which we expect to generate several stalled scenes. And the last configuration has a reduced number of User agents for detecting unsatisfiability of requests. Parameters for the three configurations, labeled Low, High and Critical respectively, are shown in Table 1.

3.5 Results

Using the configurations given above we ran experiment rounds auditing 50 new information pieces in order to measure the behavior of agents and measure the performance in the Correction scene. We observed the maximal number of simultaneously stalled scenes, i.e. scenes in hold due to a request for agents, and calculated the average conclusion time for these scenes. Additionally, we observed the maximum

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Low</th>
<th>High</th>
<th>Critical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feeding rate</td>
<td>30 sec.</td>
<td>10 sec.</td>
<td>10 sec.</td>
</tr>
<tr>
<td>Expert revision</td>
<td>2-5 sec.</td>
<td>2-5 sec.</td>
<td>2-5 sec.</td>
</tr>
<tr>
<td>Author revision</td>
<td>20-25 sec.</td>
<td>20-25 sec.</td>
<td>20-25 sec.</td>
</tr>
<tr>
<td>MaxAgs(Carrier)</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>MaxAgs(User)</td>
<td>10</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1: Experiment configurations.
number of concurrent Carrier and Users agents, as well as the maximum simultaneous number of User agents playing the Expert or Author role; recall that experts and authors use the User agent class for participating in the system. Results for the three configurations are shown in Table 2.

The first configuration showed only one Correction scene most of the time, and reached the maximum of two at some point of the simulation. The number of simultaneously stalled Correction scenes and Carrier agents is the same as long as Carrier agents are in charge of creating the Correction scene. Only one expert and one author were active in the system at the same time, authors and expert were released once the correction scene finished and were instantiated again when a new Correction scene was generated.

In the second configuration the reduction on the feeding rate produced more stalled scenes and a higher utilization of agents. The maximal number of Carrier and User agents was reached. This time the maximum number of stalled scenes didn’t match the maximal number of Carriers agents because they were busy participating in other scenes of the auditing process. The average conclusion time for scenes was doubled as long as busy experts kept in hold at most two scenes meanwhile they were attending another scene. Figure 6 shows the behavior of the population of Carrier and User agents, broke down on Expert and Author roles, for this configuration.

In the third configuration, after approximately twelve successful evaluations the entire system stalled. At this point we observed the five user agents playing the role of Expert leaving no space for authors. Ten scenes were stalled, five of which had an Expert agent and the other five had a single Carrier agent waiting. In this case the dispatcher agent was not capable of determining the unsatisfiability of the requests for authors as long as it was expecting that some User agent left the institution for instantiating an agent to play the author role. The rest of the scenes made use of the five available Expert agents not allowing the instantiation of a new User agent for playing the role of Author. All these scenes finished thanks to the timeout of 200 seconds, as can be observed in the average termination time for these scenes.

This last scenario make us conclude that it was necessary to reserve agent slots for authors in order to conclude the scenes satisfactorily. Even when the participation of experts and authors is not assured in all the scenes, we should be capable of indicating it to the dispatcher agent in order to prevent the deadlock.

Table 2: Experimental results.

<table>
<thead>
<tr>
<th>Observation</th>
<th>Low</th>
<th>High</th>
<th>Critical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. stalled scenes</td>
<td>2</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>Avg. scene time</td>
<td>30 sec.</td>
<td>61 sec.</td>
<td>197 sec.</td>
</tr>
<tr>
<td>Max. active Carriers</td>
<td>2</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Max. active Users</td>
<td>2</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Max. active Experts</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Max. active Authors</td>
<td>1</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

5.5. Future Work

The request protocol can be extended to deal with future agent invitation and not just current invitations. By so doing, we could prevent the deadlock of concurrent scenes. Another option would be developing an algorithm for pruning the directed cyclic graph representing an EI protocol or scene. That would produce a reduced version of the protocol when agents for certain role are missing. A pruned protocol that doesn’t reach the final state would indicate an unsatisfiable scene execution. For instance, a Correction protocol on which the participation of Corrector, Expert and Author agents is pruned, could be detected a priori as unsuccessful.

An institutional model of public information for scenes and agents can be used to improve our proposal. Public
information of the scene and the participants may be used by the D-Agent to narrow the announcement task, and by invited agents to calculate their bids for participating in a scene. For example, a Corrector agent that knows a rule for correcting inconsistencies of a single type, should be directed only to scenes where an inconsistency of that type is being corrected.

Agent descriptions formalized through a Description Logics [3] system would allow the generation of agent profiles describing the properties that potential participant agents should have. For instance, knowing that there are three auditing rules for the repository and that every Auditor agent can only handle one single rule, the request for auditor agents for all the type of auditing rules would generate three agent profiles, one for each rule. An instance of each profile would be enough for assuring a complete auditing of each new record.

6. ACKNOWLEDGMENTS

This paper was partially funded by the Spanish Ministry of Science and Innovation AT (CSD2007-0022, INGENIO 2010) and EVE (TIN2009-14702-C02-01), by the Generalitat de Catalunya 2009-SGR-1434, by the Mexican Council for Science and Technology and by the Tecnologico de Monterrey.

7. REFERENCES


Figure 6: Agent populations on the high-demand configuration.